Metatheory of Verification Calculi in LEGO

To What Extent Does Syntax Matter?

Thomas Kleymann

LFCS Edinburgh, King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, Scotland

Abstract. Investigating soundness and completeness of verification calculi for imperative programming languages is a challenging task. Incorrect results have been published in the past. We take advantage of the computer-aided proof tool LEGO to interactively establish soundness and completeness of both Hoare Logic and the operation decomposition rules of the Vienna Development Method with respect to operational semantics. We deal with parameterless recursive procedures and local variables in the context of total correctness.

In this paper, we discuss in detail the role of representations for expressions, assertions and verification calculi. To what extent is syntax relevant? One needs to carefully select an appropriate level of detail in the formalisation in order to achieve one's objectives.

1 Introduction

We have taken advantage of the LEGO system (Lego 1998) to produce machine-checked soundness and completeness proofs for Hoare Logic and the operation decomposition rules of the Vienna Development Method (VDM). Our imperative programming language includes (parameterless) recursive procedures and local variables. We consider static binding and total correctness. This is one of the largest developments in LEGO to date. Building on a comprehensive library it additionally consists of more than 800 definitions, lemmata and theorems.

Our message to the designers and researchers of verification calculi is that conducting computer-aided soundness and completeness proofs is both a feasible and profitable task. Our fundamental contribution has been to highlight the role of auxiliary variables in Hoare Logic. Usually, assertions are interpreted as *predicates on states* where free variables denote the value of program variables in a specific state. Variables for which no counterpart appears as a program variable in the program under consideration then take on the role of auxiliary variables. They are required to relate the value of program variables in *different* states.

Our view of assertions emphasises the pragmatic importance of auxiliary variables. We have followed a proposal by Apt & Meertens (1980) to consider assertions as *relations on states and auxiliary variables*. Furthermore, we stipulate a new structural rule to adjust auxiliary variables when strengthening preconditions and weakening postconditions. This rule is stronger than all previously suggested structural rules, including Hoare's (1969) consequence rule and rules of adaptation. As a direct consequence of the new treatment of auxiliary variables,

- we were able to show that Sokołowski's (1977) calculus for recursive procedures is sound and complete if one replaces Hoare's rule of consequence with ours. In particular, none of the other structural rules introduced by Apt (1981) (which lead to a complete but unsound system) are required.
- We have clarified the relationship between Hoare Logic and its variant VDM. We were able to show that, contrary to common belief, VDM is more restrictive than Hoare Logic in that every derivation in VDM can be naturally embedded in Hoare Logic.

Deep versus Shallow Embedding. Traditionally, one defines syntax for expressions and relative to this setup, one characterises syntax of a programming language and syntax of an assertion language. Then, one describes the meaning of every syntactic construct. This approach is known as *deep embedding*. Alternatively one may shortcut this process and identify the syntactic representation with its denotation. This technique is known as *shallow embedding*.

Related Work. The pioneering work on machine-checked soundness for Hoare Logic by Gordon (1989) rests entirely on shallow embedding. Homeier (1995) extends the soundness proof to a setting with mutually recursive procedures. His encoding is based exclusively on deep embedding. Nipkow (1998) has been the first to conduct a machine-checked *completeness* proof for Hoare Logic dealing with simple imperative programs in the context of partial correctness. This contains a mixture of shallow and deep embedding. Using similar representation techniques we have extended this work to recursive procedures and local variables.

1.1 To What Extent Does Syntax Matter?

Before deciding on the embedding technique, one ought to clarify the objectives of the machine-assisted development. This induces the level of detail in which one needs to analyse involved concepts. One of the central issues in formalising metatheory is to what extent syntax needs to be formalised. Technically, one has a choice of deep versus shallow embedding.

A shallow embedding cuts down the work load and is therefore, at least for machine-checked developments, often the preferred approach. The drawbacks of shallow embedding are that

1. one can not exploit the inductive (syntactic) structure to prove properties.
2. The representation of concrete examples is often more difficult to comprehend.

As the main contribution of this paper, we clarify the role of deep versus shallow embedding. In the setting of Hoare Logic, the choice of the level of embedding has a major influence in the work involved in setting up an appropriate theory of substitutions. One needs substitutions on states, expressions and assertions. With a shallow embedding of expressions and assertions, substitutions can be expressed in terms of substituting the state space.

We have investigated the metatheory of verification calculi. It was *not* our aim to show that a proof tool such as the LEGO system is suitable to verify concrete programs. Therefore, the second drawback was of little concern to us. Our strategy has been to employ a shallow embedding whenever possible.

However, one needs to pursue soundness by induction on the structure of programs whereas completeness is conducted by induction of the derivation of correctness formulae. Hence, in light of the first drawback of a shallow embedding, one needs to insist on a deep embedding for programs and the notion of deriving correctness formulae. The main benefit of employing a shallow embedding for investigating the metatheory of verification calculi are that

- we did not have to worry at all about substitutions in assertions; an otherwise daunting prospect (Mason 1987, Homeier 1995).

– Completeness can only be established for an assertion language which is sufficiently expressive to denote all intermediate properties such as invariants. Employing a deep embedding of assertions, one would need to additionally explicitly construct syntactic representations for all possible intermediate assertions.

1.2 Overview

The outline of this paper is as follows. We first formalise the notion of a state space. We then sketch our embedding of expression, assertions and imperative programs. In Sect. 7 we discuss semantics and derivability of Hoare Logic. We motivate new rules for loops and adjusting auxiliary variables. We argue that in investigating soundness and completeness of verification calculi, one should gloss over the syntactic details of expressions and assertions. Formalising substitutions is irrelevant. We will show in Sect. 6.3 that, at least for simple imperative programs, in the soundness and completeness proof, one does not need to appeal to *any* property of a substitution function.

In Sect. 7 we show that the metatheory for verification calculi dealing with local variables is more subtle. Not only is it essential to have an adequate substitution function (on the level of states), it is also necessary to employ an *extensional* notion of equality. This requires some attention, as type-theoretic systems such as Coq and LEGO are tailored to an *intensional* type theory. The case of VDM is similar and not covered in this paper.

2 The State Space – An Example of a Dependent Type Space

The state space records the value of every program variable. Let **VAR** be the type of program variables. Previous mechanisations only consider the case of a single data type, namely natural numbers. In a type-theoretic setting, it seems natural to investigate multiple sorts. We identify the universe of data types with the universe of all types expressible in LEGO. The type of variables can be declared by providing a function

$$\mathrm{sort} : \mathbf{VAR} \to \mathtt{Type} \ .$$

A state σ for a type environment sort is a function mapping program variables x to values of type $\mathrm{sort}(x)$. The state space itself is therefore a dependent function space:

Definition 1 (State Space). $\Sigma \stackrel{\mathrm{def}}{=} \prod x : \mathbf{VAR} \cdot \mathrm{sort}(x)$

We have implemented a substitution operation on *dependent* functions which satisfies the specification

$$\sigma[x \mapsto t]\,(y) = \begin{cases} t & \text{if } x = y, \\ \sigma(x) & \text{otherwise.} \end{cases} \tag{1}$$

This requires quite sophisticated type theory. See (Kleymann 1998) for details.

3 Expressions

Boolean expressions occur in loops and conditional statements. Other types of expressions depend on the data types expressible in the language and occur both as subexpressions of boolean expressions and in the assignment statement. One may define the syntax of expressions by a BNF grammar.

Example 1 (Syntax of Expression). Homeier & Martin (1996) define two classes of expressions

$$\begin{aligned} e &::= n \mid x \mid {++}x \mid e_1 + e_2 \mid e_1 - e_2 \\ b &::= e_1 = e_2 \mid e_1 < e_2 \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid \neg b \end{aligned}$$

We will only consider expressions without side-effects[1] and do not deal with the expression $++x$. The semantics can thus be easily fixed denotationally and is determined by an interpretation function I and a state σ. An interpretation determines the value of constants such as 0, $+$, $\wedge$ and (free) variables in expressions and logical formulae e.g., $[\![x]\!](I(\sigma)) \stackrel{\mathrm{def}}{=} I\big(\sigma(x)\big)$. Whenever we come across a boolean expression in a loop or a conditional statement, we are only interested in the value it evaluates to, **true** or **false**. Similarly, in an assignment, we treat evaluation of the

[1] Such a strict distinction between expressions and commands is one of the fundamental principles underlying idealised Algol (Reynolds 1982).

expression as atomic, merely a value depending on the state space. We are not interested in syntactic properties such as whether one expression is a subterm of another expression. Ignoring the syntax of expressions paves the way towards a reasonable level of abstraction when investigating properties of verification calculi for imperative programs without side-effects.

Furthermore, we are only interested in the standard interpretation[2] of constants. Hence, the state space alone determines the semantics. We only consider expressions at this semantic level:

Definition 2 (Expressions – Shallow Embedding). *Given an arbitrary type T, we represent expressions by*

$$\text{expression}(T : \texttt{Type}) \stackrel{\text{def}}{=} \Sigma \to T \quad .$$

Let $e : \text{expression}(T)$ *be any expression. Its evaluation depends on a concrete snapshot of the state space* $\sigma : \Sigma$ *We define* $\text{eval}(\sigma)(e) \stackrel{\text{def}}{=} e(\sigma)$.

A benefit of adopting shallow embedding is that we do not have to worry about formalising the syntax in a logical framework. Working on the metatheory, one never encounters a concrete expression! Moreover, substitutions are much easier to deal with at the semantic level. It can be defined in terms of updating states:

Definition 3 (Updating Expressions – Shallow Embedding).

$$e\,[x \mapsto t]\,(\sigma) \stackrel{\text{def}}{=} e(\sigma\,[x \mapsto t])$$

In a deep embedding, one would need to define an interpretation and substitution function by induction on the structure of expressions and *prove* the substitution lemma

$$[\![e\,[x \mapsto t]]\!](\sigma) = [\![e]\!](\sigma\,[x \mapsto t]) \quad .$$

An advantage of deep embedding is that, for concrete expressions, substitutions are more palatable. Consider the syntactic substitution

$$(x * y)\,[x \mapsto 3] \quad .$$

[2] This not only simplifies the encoding, it also avoids the problematic issue of how to axiomatise the class of acceptable interpretations. In particular, incompleteness results of Hoare Logic e.g., (Clarke Jr. 1979), exploit setups with *non-standard* interpretations.

Due to the recursive definition of updating, this should reduce to $3*y$. In the shallow embedding, we would instead have

$$\lambda\sigma\cdot\left(\underbrace{\left(\lambda\sigma\cdot\sigma(x)*\sigma(y)\right)}_{[\![x*y]\!]}\left(\sigma\left[x\mapsto 3\right]\right)\right) \tag{2}$$

which (β-)reduces to $\lambda\sigma\cdot\left(\sigma\left[x\mapsto 3\right](x)*\sigma\left[x\mapsto 3\right](y)\right)$. This is equivalent to

$$\lambda\sigma\cdot 3*\sigma(y) \tag{3}$$

Unfortunately, the LEGO system offers little support for reducing (2) to (3). In concrete examples, this leads to excessively large proof obligations. Computer-aided verification becomes unfeasible[3].

4 Assertions

Traditionally, assertions are considered to be simply formulae of first-order logic, which are interpreted in the usual way, except that the value of variables is determined by a state. Semantically, from a type-theoretic point of view, assertions are the particular class of expressions over propositions i.e., expression(`Prop`). Instead of first-order logic it is convenient to exploit the native logic of the theorem prover. This encoding has been adopted in (Gordon 1989, Nipkow 1998).

Our novel approach to Hoare Logic has been to give a more rigorous treatment of auxiliary variables. They are required at the level of specification to relate the value of variables in different states as assertions may otherwise only relate the value of program variables in a *single* state.

At the syntactic level, one would need to (formally) distinguish between program variables and auxiliary variables. One could for example enforce that program variables have to start with a lower-case letter, whereas auxiliary variables must start with an upper-case letter. To be well-formed, programs may only refer to program variables.

[3] In a verification of the recursive algorithm Quicksort we had not manually intervened in reducing substitutions. For the correctness proof, LEGO had to run for more than 37 hours requiring more than 80MB on a SUN SPARC station 20 with sufficient physical memory to avoid swapping. In comparison, on the same architecture, the completeness proof for Hoare Logic dealing with recursive procedures and local variables could be dealt with in less than 15 minutes requiring less than 25MB. In both cases, we started LEGO in the empty environment.

Semantically, program variables are, as before, interpreted according to the state space. However, auxiliary variables are interpreted freely. Let T be the domain of this interpretation.

Definition 4 (Assertions – Shallow Embedding).

$$\text{Assertion}(T : \texttt{Type}) \stackrel{\text{def}}{=} (T \times \Sigma) \to \texttt{Prop}$$

Example 2. Let $T = \{X,Y\} \to \texttt{int}$. Relative to an interpretation $Z : T$ and a state σ, we interpret $[\![0 \leq y \wedge x = X \wedge y = Y]\!](Z,\sigma) = 0 \leq \sigma(y) \wedge \sigma(x) = Z(X) \wedge \sigma(y) = Z(Y)$.

Due to the shallow embedding we may update assertions analogue to expressions by relaying the work to updating the state space. In practice we only need to update the value of program variables but not auxiliary variables. Let p be an assertion.

Definition 5 (Updating Assertions – Shallow Embedding).

$$p\,[x \mapsto t]\,(Z,\sigma) \stackrel{\text{def}}{=} p(Z,\sigma\,[x \mapsto t]) \ \ .$$

Analogue to expressions, in a deep embedding, one would need to additionally represent syntax for assertions, define an interpretation and a syntactic substitution function. Then, one would need to *prove* the substitution lemma

$$[\![p\,[x \mapsto t]]\!](Z,\sigma) = [\![p]\!](Z,\sigma\,[x \mapsto t]) \ \ .$$

5 Imperative Programs

A shallow embedding of a non-trivial imperative programs is problematic for a strongly-types proof system such as LEGO. An imperative program S may not terminate. Hence, the denotation of S is a *partial* functions on the state space, $[\![S]\!] : \Sigma \rightharpoonup \Sigma$. However, LEGO only supports total functions. To avoid partiality, one may move to relational denotational semantics $[\![S]\!] : (\Sigma \times \Sigma) \to \texttt{bool}$, see (Gordon 1989) for an example.

In any case, to formally prove soundness within a logical framework, one needs to pursue induction on the structure of programs. Thus, one has to select a deep embedding strategy for the imperative programming language. For the purpose of this section, we consider a (very) simple imperative programming language consisting of assignments and loops.

Definition 6 (Syntax of Imperative Programs – Deep Embedding). *Imperative programs* $S : \texttt{prog}$ *are defined by the BNF grammar*

$$S ::= x := t \mid \textbf{while } b \textbf{ do } S$$

where $x : \textbf{VAR}$, $t : \text{expression}(\text{sort}(x))$ *and* $b : \text{expression}(\texttt{bool})$.

We employ structural operational semantics which provides a clean way to specify the effect of each language constructor in an arbitrary state. It relates a program with its initial and final state.

Definition 7 (Structural Operational Semantics). *The operational semantics is defined as the least relation* $. \xrightarrow{\;\;\;.\;\;\;} . \subseteq \Sigma \times \texttt{prog} \times \Sigma$ *satisfying*

$$\sigma \xrightarrow{x:=t} \sigma[x \mapsto \text{eval}(\sigma)(t)] \tag{4}$$

$$\sigma \xrightarrow{\textbf{while } b \textbf{ do } S} \sigma \qquad \textit{provided } \text{eval}(\sigma)(b) = \textbf{false} \;.$$

$$\frac{\sigma \xrightarrow{S} \eta \quad \eta \xrightarrow{\textbf{while } b \textbf{ do } S} \tau}{\sigma \xrightarrow{\textbf{while } b \textbf{ do } S} \tau} \qquad \textit{provided } \text{eval}(\sigma)(b) = \textbf{true} \;.$$

Intuitively, $\sigma \xrightarrow{S} \tau$ denotes that the program S when invoked in the state σ will terminate in the state τ.

6 Semantics and Derivability of Hoare Logic

Hoare Logic is a verification calculus for deriving correctness formulae of the form $\{p\}\, S\, \{q\}$ for assertions p, q and programs S. We consider total correctness. Intuitively $\{p\}\, S\, \{q\}$ specifies that, provides S is executed in a state such that the precondition p holds, it terminates in a state τ where the postcondition q is satisfied. One distinguishes between the semantics of a correctness formulae $\models_{\text{Hoare}} \{p\}\, S\, \{q\}$ (which formalises the above intuition) and the notion of deriving a correctness formulae $\vdash_{\text{Hoare}} \{p\}\, S\, \{q\}$ (which is employed in order to verify concrete programs). We refer the reader to (Cousot 1990) for a comprehensive overview of Hoare Logic.

Definition 8 (Semantics of Hoare Logic). *Parametrised by an arbitrary type T, let* $\models_{\text{Hoare}} \{.\}\ .\ \{.\} \subseteq \text{Assertion}(T) \times \texttt{prog} \times \text{Assertion}(T)$ *be a new judgement defined in terms of the operational semantics*

$$\models_{\text{Hoare}} \{p\}\ S\ \{q\} \stackrel{\text{def}}{=} \forall Z \cdot \forall \sigma \cdot p(Z,\sigma) \Rightarrow \exists \tau \cdot \sigma \xrightarrow{\quad S \quad} \tau \wedge q(Z,\tau) \ .$$

Based on work of Floyd (1967), Hoare (1969) proposed a verification calculus for partial correctness, now referred to as Hoare Logic. For every constructor of the imperative programming language, Hoare Logic provides a rule which allows one to decompose a program. The precondition of the assignment axiom

$$\{p[x \mapsto t]\}\ x := t\ \{p\}$$

is, at least for simple imperative programs, the sole reason for having to bother about updating assertions!

Programs mentioned in the premisses are strict subprograms of the programs mentioned in the conclusions. Unlike the operational semantics, this also holds for loops.

$$\frac{\{p \wedge b\}\ S\ \{p\}}{\{p\}\ \textbf{while}\ b\ \textbf{do}\ S\ \{p \wedge \neg b\}} \tag{5}$$

One also needs a structural rule to weaken the precondition and strengthen the postcondition is a proof obligation.This is particularly useful when one wants to apply the rule for loops as the precondition must remain invariant with respect to the body of the loop.

$$\frac{\{p_1\}\ S\ \{q_1\}}{\{p\}\ S\ \{q\}} \qquad \text{provided } p \Rightarrow p_1 \text{ and } q_1 \Rightarrow q. \tag{6}$$

6.1 Total Correctness

To ensure termination, the rule for loops (5) needs to be modified. We introduce a termination measure $u : \text{expression}(W)$ for some well-founded structure $(W, <)$ which is decreased whenever the body is executed:

$$\frac{\forall t : W \cdot \{p \wedge b \wedge u = t\}\ S\ \{p \wedge u < t\}}{\{p\}\ \textbf{while}\ b\ \textbf{do}\ S\ \{p \wedge \neg b\}}$$

A similar rule for verification calculi where postconditions may explicitly refer to the value of program variables in the initial state e.g., VDM, has been put forward by Manna & Pnueli (1974). Variants of this rule tailored for $W = \texttt{nat}$ (Harel 1980) or $W = \texttt{int}$ (Apt & Olderog 1991) have also been published previously. We prefer the well-founded version, because it simplifies the completeness proof without any impact on the soundness proof (Kleymann 1998). It is well known that in practice, it is often easier to reason about termination using well-founded sets rather than being restricted to natural numbers (Dershowitz & Manna 1979).

6.2 Auxiliary Variables

Furthermore, we have strengthened the rule of consequence (6) so that one may adjust auxiliary variables when strengthening preconditions and weakening postconditions. Let x be a list of all program variables and Z be a list of all auxiliary variable.

$$\frac{\{p_1\}\; S\; \{q_1\}}{\{p\}\; S\; \{q\}}$$

provided $\forall Z \cdot \forall x \cdot \exists Z_1 \cdot \big(p \Rightarrow (p_1\,[Z \mapsto Z_1])\big) \wedge \big(\forall x \cdot (q_1\,[Z \mapsto Z_1]) \Rightarrow q\big)$

Example 3 (Auxiliary Variables). With this rule (but not Hoare's (6)), the two correctness formulae

$$\{X = x\}\; S\; \{X = x\}$$

and

$$\{X = x+1\}\; S\; \{X = x+1\} \quad ,$$

where all variables denote integer values and X is an auxiliary variable which does not occur in S, are interderivable.

The new rule of consequence plays a crucial role in deriving the Most General Formlua (MGF), the key theorem to establish completeness for Hoare Logic dealing with recursive procedures (Schreiber 1997, Kleymann 1998).

However, since LEGO is a constructive framework, the side-condition is too restrictive. Instead, one needs to consider (the intuitionistically weaker)

$$\forall Z \cdot \forall x \cdot p \Rightarrow \Big(\exists Z_1 \cdot (p_1\,[Z \mapsto Z_1]) \wedge \big(\forall x \cdot (q_1\,[Z \mapsto Z_1]) \Rightarrow q\big)\Big) \quad .$$

Definition 9 (Derivability of Hoare Logic – Deep Embedding). *A verification calculus for Hoare Logic is defined as the least relation*

$$\vdash_{\text{Hoare}} \{.\}\ .\ \{.\} \subseteq \text{Assertion}(T) \times \mathtt{prog} \times \text{Assertion}(T)$$

indexed by an arbitrary type T such that

$$\vdash_{\text{Hoare}} \big\{\lambda(Z,\sigma)\cdot p\big(Z,\sigma[x \mapsto \text{eval}(\sigma)(t)]\big)\big\}\ x := t\ \{p\} \tag{7}$$

$$\frac{\begin{array}{c}\forall t : W \cdot \vdash_{\text{Hoare}} \{\lambda(Z,\sigma)\cdot p(Z,\sigma) \wedge \text{eval}(\sigma)(b) = \textbf{true} \wedge \text{eval}(\sigma)(u) = t\} \\ S \\ \{\lambda(Z,\tau)\cdot p(Z,\tau) \wedge \text{eval}(\tau)(u) < t\}\end{array}}{\vdash_{\text{Hoare}} \{p\}\ \textbf{while}\ b\ \textbf{do}\ S\ \{\lambda(Z,\tau)\cdot p(Z,\tau) \wedge \text{eval}(\tau)(b) = \textbf{false}\}}$$

where $(W,<)$ is well-founded.

$$\frac{\vdash_{\text{Hoare}} \{p_1\}\ S\ \{q_1\}}{\vdash_{\text{Hoare}} \{p\}\ S\ \{q\}}$$

$$\textit{provided}\ \forall Z \cdot \forall \sigma \cdot p(Z,\sigma) \Rightarrow \Big(\exists Z_1 \cdot p_1(Z_1,\sigma) \wedge \big(\forall \tau \cdot q_1(Z_1,\tau) \Rightarrow q(Z,\tau)\big)\Big). \tag{8}$$

6.3 Soundness

Soundness is essential. If a system is unsound, deriving a property for a particular program within the formal system does not guarantee that the program actually fulfils the property. Formally, one needs to show that whenever a correctness formulae $\vdash_{\text{Hoare}} \{p\}\ S\ \{q\}$ is derivable, the proposition $\models_{\text{Hoare}} \{p\}\ S\ \{q\}$ holds. Soundness is best pursued by induction on the derivation of the correctness formula. For the discussion of deep versus shallow embedding, the case of an assignment is of peculiar interest.

Lemma 1 (Soundness of Assignment Axiom).

$$\models_{\text{Hoare}} \big\{\lambda(Z,\sigma)\cdot p\big(Z,\sigma[x \mapsto \text{eval}(\sigma)(t)]\big)\big\}\ x := t\ \{p\}$$

Proof: Expanding the definition of $\models_{\text{Hoare}}$, given Z, σ, one needs to establish

$$p(Z,\sigma[x \mapsto \text{eval}(\sigma)(t)]) \Rightarrow \exists\tau \cdot \sigma \xrightarrow{x:=t} \tau \wedge p(Z,\tau) \enspace .$$

The operational semantics uniquely determines the final state τ. Appealing to the axiom (4), it suffices to show

$$p(Z,\sigma[x \mapsto \text{eval}(\sigma)(t)]) \Rightarrow p(Z,\sigma[x \mapsto \text{eval}(\sigma)(t)]) \enspace .$$

■

As expected, due to a shallow embedding, we only have one notion of substitution (on the level of states). But perhaps surprisingly, soundness holds *regardless* of the details of the actual substitution function.

If one is only interested in establish soundness (and not completeness), there is no need for any deep embeddings. Induction on the structure of programs is not required. Hence, there is no need for a deep embedding of imperative programs e.g., Gordon (1989) represents programs by their relational denotational semantics.

A Shallow Embedding of Hoare Logic. Moreover, if one externalises the induction of the soundness proof to the meta-level as opposed to the proof tool, one can give a shallow embedding for Hoare Logic. Without a notion of derivability as given in Definition 9, soundness can be established by showing that axioms are valid with respect to $\models_{\text{Hoare}}$ and that all rules preserve soundness. This approach has been pursued by (Gordon 1989, Homeier 1995, Homeier & Martin 1996, Norrish 1996).

One must however be clear about the limitations of this approach. For example, Homeier & Martin (1996) erroneously claim that the soundness of a (complete) Verification Condition Generator (VCG) has been established by appealing to the axioms and rules of an (incomplete) presentation of Hoare Logic[4]. But since they employ a *shallow* embedding of Hoare Logic, correctness of the VCG has instead been established by appealing to the definition of *operational* semantics.

[4] A consequence rule is missing. Thus, one can e.g. not derive $\{x = 1\}$ **skip** $\{\textbf{true}\}$.

6.4 Completeness

In an incomplete formal system, one may only verify a strict subset of all true formulae. A naive definition of completeness is bound to fail in the context of verification calculi. On the one hand, if the chosen underlying logical language is too weak, e.g., pure first-order logic together with the boolean constants **false** and **true**, some intermediate assertions cannot be expressed. Hence, derivations cannot be completed. On the other hand, if the logical language is too strong, e.g. Peano Arithmetic, it itself is already incomplete and the verification calculus inherits incompleteness.

To avoid this problem, Cook (1978) has proposed that one investigates *relative completeness* in an attempt to separate the reasoning about programs from the reasoning about the underlying logical language. One only considers expressive first-order logics. Furthermore, rules of the verification calculus may be applied in a derivation if the logical side-condition is valid rather than derivable. In particular, completeness nolonger compares a proof-theoretic with a model-theoretic account.

In practice, achieving relative completeness of verification calculi is highly desirable. In logic, finding valid formulae which can not be derived is often somewhat esoteric. A different story has to be told for the notion of relative completeness in verification calculi e.g., in Sokołowski's (1977) calculus, it is very difficult to come up with any non-contrived correctness formula of a recursive procedure which can be derived!

In a machine-checked development, it is convenient to interpret Cook's proposal by employing the native (expresssive) logic of the theorem prover to interpret assertions. A shallow embedding of assertions automatically blurs the model and proof-theoretic aspect of assertions. As an important aspect in the completeness proof, one needs to be able to formulate an assertion which expresses the weakest precondition relative to an arbitrary program and postcondition. With a shallow embedding, this is straightforward:

Definition 10 (Weakest Precondition – Shallow Embedding).

$$\mathrm{wp}(S,q)(Z,\sigma) \stackrel{\text{def}}{=} \exists\tau \cdot \sigma \xrightarrow{\quad S \quad} \tau \wedge q(Z,\tau) \ .$$

With a deep embedding of assertions, one would have to derive a syntactic representation which denotes the weakest precondition. This is considerably more challenging[5].

One may prove completeness directly by induction on the structure of S. Instead, we follow a technique developed by Gorelick (1975), which, previously, has only been applied to the scenario of Hoare Logic dealing with recursive procedures :

1. By induction on the structure of an arbitrary program S, one establishes that a specific correctness formula $\mathrm{MGF}_{\mathrm{Hoare}}(S)$ is derivable in the verification calculus.
2. Given the assumption $\models_{\mathrm{Hoare}} \{p\}\ S\ \{q\}$, one may derive

$$\vdash_{\mathrm{Hoare}} \{p\}\ S\ \{q\}$$

by applying structural rules to $\vdash_{\mathrm{Hoare}} \mathrm{MGF}_{\mathrm{Hoare}}(S)$. All side-conditions which arise will be dealt with by the assumption.

In other words, instead of directly deriving

$$\models_{\mathrm{Hoare}} \{p\}\ S\ \{q\} \Rightarrow \vdash_{\mathrm{Hoare}} \{p\}\ S\ \{q\} \quad ,$$

one considers the stronger property $\vdash_{\mathrm{Hoare}} \mathrm{MGF}_{\mathrm{Hoare}}(S)$ for which induction goes through more easily. In particular, the direct proof can not be applied when one considers recursive procedures, because the induction hypotheses are not strong enough.

The proposition $\vdash_{\mathrm{Hoare}} \mathrm{MGF}_{\mathrm{Hoare}}(S)$ asserts that, provided that one only considers input states in which the program S terminates, one may *derive* a correctness formula in which the postcondition relates all inputs with the appropriate outputs according to the underlying operational semantics of the programming language. At the semantic level, $\models_{\mathrm{Hoare}} \mathrm{MGF}_{\mathrm{Hoare}}(S)$ holds trivially.

Definition 11 (MGF – Shallow Embedding).

$$\mathrm{MGF}_{\mathrm{Hoare}}(S) \stackrel{\mathrm{def}}{=} \left\{\lambda(Z,\sigma)\cdot\sigma \xrightarrow{\qquad S \qquad} Z\right\}\ S\ \{\lambda(Z,\tau)\cdot Z=\tau\}$$

[5] If the assertion language is Peano Arithmetic, this construction is not for the faint-hearted as one has to work on the level of Gödel numbers (de Bakker 1980).

Notice that the precondition is equivalent to the the weakest precondition relative to the postcondition $\lambda(Z,\tau)\cdot Z=\tau$.

Analogue to the proof of soundness, in deriving the MGF for assignment, one again encounters the phenomenon that the details of the substitution function are irrelevant.

7 Extensional Equality and Local Variables

In the previous section, we have seen that, for soundness (and completeness), details of substitutions can be neglected. Catering for local initialised variables **new** $x:=t$ **in** S is however more demanding, because one needs to reinstate the previous value of x after the block. Based on an idea by Sieber (1981), Olderog (1981) captures the semantics of blocks by

$$\frac{\sigma[x\mapsto \mathrm{eval}(\sigma)(t)] \xrightarrow{S} \tau}{\sigma \xrightarrow{\mathbf{new}\ x:=t\ \mathbf{in}\ S} \tau[x\mapsto\sigma(x)]}\ . \tag{9}$$

To verify programs containing blocks, we have proposed the rule

$$\frac{\forall v\cdot\{p[x\mapsto v]\wedge x=t[x\mapsto v]\}\ S\ \{q[x\mapsto v]\}}{\{p\}\ \mathbf{new}\ x:=t\ \mathbf{in}\ S\ \{q\}}\ .$$

Taking into account a shallow embedding of assertions, this corresponds formally to

$$\frac{\begin{array}{c}\forall v\cdot\vdash_{\mathrm{Hoare}}\{\lambda(Z,\sigma)\cdot p(Z,\sigma[x\mapsto v])\wedge\sigma(x)=\mathrm{eval}(\sigma[x\mapsto v])(t)\}\\ S\\ \{q[x\mapsto v]\}\end{array}}{\{p\}\ \mathbf{new}\ x:=t\ \mathbf{in}\ S\ \{q\}} \tag{10}$$

It is an improvement over Apt's (1981) version in that it deals with initialised blocks. Furthermore, no side-conditions are required[6]. In the soundness and completeness proof, we need to appeal to the following two extensional properties of substitutions:

$$\sigma[x\mapsto\sigma(x)]=\sigma \tag{11}$$

$$\sigma[x\mapsto t_1][x\mapsto t_2]=\sigma[x\mapsto t_2] \tag{12}$$

We restrict our attention to the crucial step of the completeness proof:

[6] Scoping of the implicitly universally quantified p, S and q ensures that $v\notin \mathit{free}(p,S,q)$.

Lemma 2 (MGF for Blocks). *Whenever one can derive*

$$\vdash_{\text{Hoare}} \text{MGF}_{\text{Hoare}}(S) \ ,$$

one may also establish

$$\vdash_{\text{Hoare}} \text{MGF}_{\text{Hoare}}(\mathbf{new}\ x := t\ \mathbf{in}\ S) \ .$$

Proof: Given an arbitrary v : sort(x), we apply the (stronger) rule of consequence (8) to the hypothesis $\vdash_{\text{Hoare}} \text{MGF}_{\text{Hoare}}(S)$ in order to derive

$$\vdash_{\text{Hoare}} \left\{\lambda(Z,\sigma)\cdot\sigma[x\mapsto v] \xrightarrow{\mathbf{new}\ x := t\ \mathbf{in}\ S} Z \wedge \sigma(x) = \text{eval}(\sigma[x\mapsto v])(t)\right\}$$
$$S$$
$$\{\lambda(Z,\tau)\cdot Z = \tau[x\mapsto v]\} \tag{13}$$

From (13), the rule for blocks (10) renders the proof obligation. As a side-condition, given states Z and σ such that

$$\sigma[x\mapsto v] \xrightarrow{\mathbf{new}\ x := t\ \mathbf{in}\ S} Z \tag{14}$$

$$\sigma(x) = \text{eval}(\sigma[x\mapsto v])(t) \tag{15}$$

we have to find a state τ such that $\sigma \xrightarrow{\quad S \quad} \tau$ and $Z = \tau[x\mapsto v]$. Inverting the derivation of (14), there must be such a state τ which satisfies

$$\sigma[x\mapsto v][x\mapsto \text{eval}(\sigma[x\mapsto v])(t)] \xrightarrow{\quad S \quad} \tau \tag{16}$$

and $Z = \tau[x\mapsto\sigma[x\mapsto v](x)]$. Courtesy of (15), the property (16) can be simplified to $\sigma[x\mapsto v][x\mapsto\sigma(x)] \xrightarrow{\quad S \quad} \tau$. To complete the proof, one needs to appeal to the substitution properties (11) and (12) and replace $\sigma[x\mapsto v][x\mapsto\sigma(x)]$ by the extensional equal function σ. ■

It follows from the specification of the update operation on states (1) that we may derive the extensional counterparts of (11) and (12)

$$\sigma[x\mapsto\sigma(x)](y) = \sigma(y)$$
$$\sigma[x\mapsto t_1][x\mapsto t_2](y) = \sigma[x\mapsto t_2](y)$$

whereas (11) and (12) themselves do not hold for the standard equality concepts such as Leibniz or Martin-Löf equality, because they distinguish between intensionally distinct functions. We therefore need to *axiomatise* extensionality (Hofmann 1995).

8 Conclusions

To prove completeness, one needs to be able to construct assertions which express semantic properties of the programming language. On paper, one usually simply assumes that the assertion language is sufficiently expressive. Both, soundness and completeness proofs can be simplified if one does not worry about the actual syntactic representation of assertions.

Moreover, a thorough treatment of syntax has the unpleasant side-effect that substantial amount of formal detail is required to deal with substitutions at the level of states, expressions and assertions. This seems redundant as far as metatheory is concerned. Specifically, for simple imperative programs, the proofs of soundness and completeness can be conducted irrespective of the chosen substitution function. Semantically, the assignment axiom in Hoare Logic simply lifts substitutions pointwise from the level of states to predicates on states.

Syntax does however matter if, instead of metatheory, one wishes to use the axioms and rules to verify concrete programs or generate verification conditions. With a shallow embedding, assertions are functions mapping states to propositions. Not only are they more difficult to comprehend than their syntactic counterpart. Without syntactic structure, the proof tool has little guidance on how to best reduce substitutions in assertions. Verifying the Quicksort algorithm based on a shallow embedding, we found that the resulting proof obligations arising from the side-condition of the rule of consequence become too large for the LEGO system to efficiently process. Having to deal with *dependent* types, type-checking involves expensive calculations.

One ought to clarify the objectives of employing a theorem prover. There are two *orthogonal* problems in verifying imperative programs.

1. Establishing soundness and completeness for verification calculi is a challenging task. Incorrect results based on doing proofs by hand have been published in the past. The metatheory relates semantics and derivability. Syntax of assertions is not an issue. In fact, the whole idea of relative completeness is to factor out the issue of semantics versus derivability of assertions.
2. Verifying concrete programs is a labour-intensive task for which computer-aided support is vital.

We feel, a reasonable approach would be to employ a shallow embedding for metatheory and a deep embedding for concrete examples. The calculus for verifying concrete programs can *informally* build on the axioms and rules investigated in the meta-theoretical analysis. Relating the two formalisations centers mostly on the issue of how expressive the assertion language is. We are somewhat sceptical whether this deserves a machine-checked proof.

But perhaps, there is an alternative. Today's proof tools are equipped with a powerful native logic e.g., LEGO supports intuitionistic higher-order logic with a rich universe of data types (Luo 1994). However, this can not be directly employed for a deep embedding because its syntax is not inductively defined at the level of the proof system. But one could consider to treat syntax at a more informal level. Specifically, based on a shallow embedding, one could employ parsing and pretty-printing of the theorem prover to convert between the internal representation and the user interface. Moreover, one could tailor the prover's tactics engine to better deal with substitutions. At the code level of the theorem prover, it is easier to implement a suitable substitution function for a particular class of terms.

Acknowledgements

Thanks to Martin Hofmann for helpful comments on an earlier version of this report. I would also like to acknowledge the financial support of EPSRC, the British Council (ARC Project Co-Development of Object-Oriented Programs in LEGO) and the European Commission (Marie Curie Fellowship).

References

Apt, K. R. (1981), 'Ten years of Hoare's logic: A survey – part I', *ACM Transactions on Programming Languages and Systems* **3**(4), 431–483.

Apt, K. R. & Meertens, L. G. L. T. (1980), 'Completeness with finite systems of intermediate assertions for recursive program schemes', *SIAM Journal on Computing* **9**(4), 665–671.

Apt, K. R. & Olderog, E.-R. (1991), *Verification of Sequential and Concurrent Programs*, Texts and Monographs in Computer Science, Springer, New York.

Clarke Jr., E. M. (1979), 'Programming language constructs for which it is impossible to obtain good Hoare axiom systems', *Journal of the ACM* **26**(1), 129–147.

Cook, S. A. (1978), 'Soundness and completeness of an axiom system for program verification', *SIAM Journal on Computing* **7**(1), 70–90.

Cousot, P. (1990), Methods and logics for proving programs, *in* J. van Leeuwen, ed., 'Handbook of Theoretical Computer Science', Vol. B: Formal Models and Semantics, Elsevier, chapter 15, pp. 841–993.

de Bakker, J. (1980), *Mathematical Theory of Program Correctness*, Prentice Hall.

Dershowitz, N. & Manna, Z. (1979), 'Proving termination with multiset orderings', *Communications of the ACM* **22**(8), 465–475.

Floyd, R. (1967), Assigning meanings to programs, *in* J. T. Schwartz, ed., 'Proc. Symp. in Applied Mathematics', Vol. 19, pp. 19–32.

Gordon, M. J. (1989), Mechanizing programming logics in higher order logic, *in* G. Birtwhistle & P. Subrahmanyam, eds, 'Current Trends in Hardware Verification and Automated Theorem Proving (Banff, Alberta)', number 15 *in* 'Workshops in Computing', Springer-Verlag, pp. 387–439.

Gorelick, G. A. (1975), A complete axiomatic system for proving assertions about recursive and non-recursive programs, Technical Report 75, Department of Computer Science, University of Toronto.

Harel, D. (1980), 'Proving the correctness of regular deterministic programs: A unifying survey using dynamic logic', *Theoretical Computer Science* **12**, 61–81.

Hoare, C. A. R. (1969), 'An axiomatic basis for computer programming', *Communications of the ACM* **12**, 576–580.

Hofmann, M. (1995), Extensional concepts in intensional type theory, PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh.
*http://www.dcs.ed.ac.uk/lfcsreps/EXPORT/95/ECS-LFCS-95-327/

Homeier, P. V. (1995), Trustworthy Tools for Trustworthy Programs: A Mechanically Verified Verification Condition Generator for the Total Correctness of Procedures, PhD thesis, University of California, Los Angeles.
*http://www.cis.upenn.edu/~homeier/phd.html

Homeier, P. V. & Martin, D. F. (1996), Mechanical verification of mutually recursive procedures, *in* M. A. McRobbie & J. K. Slaney, eds, 'Automated Deduction – CADE-13', Vol. 1104 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, New Brunswick, NJ, USA, pp. 201–215. 13th International Conference on Automated Deduction.

Kleymann, T. (1998), Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs, PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh.
*http://www.dcs.ed.ac.uk/lfcsreps/EXPORT/98/ECS-LFCS-98-392/

Lego (1998), 'The LEGO proof assistant'.
*http://www.dcs.ed.ac.uk/home/lego

Luo, Z. (1994), *Computation and Reasoning: A Type Theory for Computer Science*, Oxford University Press.

Manna, Z. & Pnueli, A. (1974), 'Axiomatic approach to total correctness of programs', *Acta Informatica* **3**, 243–263.

Mason, I. A. (1987), Hoare's logic in the LF, Technical Report 32, Laboratory for Foundations of Computer Science, University of Edinburgh.

Nipkow, T. (1998), 'Winskel is (almost) right: Towards a mechanized semantics textbook', *Formal Aspects of Computing* . To appear.
*http://www4.informatik.tu-muenchen.de/~nipkow/pubs/winskel.html

Norrish, M. (1996), Derivation of verification rules for C from operational definitions, *in* J. von Wright, J. Grundy & J. Harrison, eds, 'Supplementary Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics: TPHOLs'96', number 1 *in* 'TUCS General Publications', Turku Centre for Computer Science, pp. 69–75.

Olderog, E.-R. (1981), 'Sound and complete Hoare-like calculi based on copy rules', *Acta Informatica* **16**, 161–197.

Reynolds, J. C. (1982), Idealized Algol and its specification logic, *in* D. Néel, ed., 'Tools & Notions for Program Construction', Cambridge University Press.

Schreiber, T. (1997), Auxiliary variables and recursive procedures, *in* M. Bidoit & M. Dauchet, eds, 'Proceedings of TAPSOFT '97', Vol. 1214 of *Lecture Notes in Computer Science*, Springer-Verlag, Lille, France, pp. 697–711.
*http://www.dcs.ed.ac.uk/home/tms/lego/tapsoft97/

Sieber, K. (1981), A new Hoare-calculus for programs with recursive parameterless procedures, Technical Report A 81/02, Fachbereich 10 – Informatik, Universität des Saarlandes, Saarbrücken.

Sokołowski, S. (1977), Total correctness for procedures, *in* J. Gruska, ed., 'Sixth Mathematical Foundations of Computer Science (Tatranská Lomnica)', Vol. 53 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 475–483.